AN 759: Using Secure Boot in Intel® Arria® 10 SoC Devices

Updated for Intel® Quartus® Prime Design Suite: **20.4**

Subscribe
Send Feedback

AN-759 | 2021.03.29
Latest document on the web: PDF | HTML

Contents

AN 759: Using Secure Boot in Intel® Arria® 10 SoC Devices....4
Prerequisites....4
References....5
Secure Boot Stages....6
Root of Trust....6
First-Stage Boot Loader (ROM)....7
Second-Stage Boot Loader....7
Third and Fourth Stages....7
Intel Arria 10 SoC Secure Boot Architecture....8
Software Image Authentication....8
Digital Signing....8
Root of Trust and Root Key....9
Authentication of the Second-Stage Boot Loader....9
Security Level Staging....10
Signed Image....11
Root Key Types....12
Root Public Key Authentication....12
Test Secure Boot Authentication....12
Programming the Secure Signing Key....13
Generating the Signing Key Pair with OpenSSL....14
Overview of the Secure Boot Flow....16
Creating a Secure Boot System....16
Software Image Encryption....18
AES Encryption and Decryption....18
Encrypting the Boot Image and Configuration File....19
Boot Image Encryption Flow....20
Programming the AES Encryption Key....20
Software Image Authentication and Encryption....21
Intel Arria 10 SoC FPGA Authentication Signing Utility....21
Secure Boot Image Tool....21
Boot Image Format Tool....22
Secure Boot Examples....23
Prerequisites....23
Creating a Signed First-Stage Boot Loader Image....23
Creating an Encrypted First-Stage Boot Loader Image....26
Creating a Signed and Encrypted First-Stage Boot Loader Image....28
Appendix A: Secure Boot Image Python Script: `alt_authtool.py`....29
Appendix B: Frequently Asked Questions....30
What are the secure configurations for HPS JTAG debug and access?....31
Can the HPS perform decryption of the boot image instead of the FPGA CSS?....31
What happens if the first stage boot ROM is unsuccessful in authenticating the second-stage boot loader?....31
Can you use the first-stage root key as the subsequent stage root key?....31
When the second-stage image is authenticated, is the image header only copied to on-chip RAM for authentication?....32
Can the AES encryption key be updated by the HPS using JTAG hosting?....32
How does U-Boot (SSBL) authenticate next stage boot images?....32

Which elliptical cryptography is used for boot image signing and authentication?..... 32
How do I generate a signing key pair?.....32
Where can I store the signing keys for second-stage boot loader authentication?.....32
What type of cryptography is used for boot image encryption and decryption?.....33
What FPGA locations are available for AES key storage?..... 33
How do I generate an AES key to encrypt a boot image?..... 33
How is secure boot defined within the Intel Arria 10 SoC product family?.....34
What security choices are available for the second-stage boot image or user software?.....34
Where is the authentication of the boot image performed?.....34
Where is decryption of the boot image performed?..... 34
How can I configure the Intel Arria 10 SoC device so that it always performs authentication or authentication and decryption?.....34
How can I program the key authentication key (KAK) into the Intel Arria 10 SoC device?.....35
How can I configure the second stage boot loader image for the correct authentication signing key type?..... 35
How do I configure the second-stage boot loader image for encryption using the pre-generated AES key?.....35
Is the ECDSA private and public key pair that is used for signing the boot image also used for authentication of the FPGA image?..... 35
Document Revision History for the AN 759: Using Secure Boot in Intel Arria 10 SoC Devices.....36

AN 759: Using Secure Boot in Intel® Arria® 10 SoC Devices

The Intel® Arria® 10 SoC device family and supported tools provide features and resources to create a secure boot system. Secure booting is essential to protect the design's intellectual property (through encryption) and prevent malicious software from running on the system (through authentication). A secure boot system establishes a chain of trust. Each piece of firmware or software is validated before running, and also validates the security signature on the next piece of software before loading it for execution.

This document provides methods and design examples for implementing an Intel Arria 10 SoC secure boot system using tools from the Intel Arria 10 SoC FPGA Authentication Signing Utility to secure the first-stage boot loader image. It shows how to generate a secure boot loader, creating and programming secure keys for image authentication and image encryption and decryption.

Note: Securing boot stages after the second-stage boot loader is outside the scope of this document and is dependent on your choice of OS and application. If the boot loader must secure subsequent boot stages (such as the operating system), you must implement a secure boot flow at the second-stage boot loader. Intel Arria 10 SoC FPGA Authentication Signing Utility does not provide any specific support for boot security beyond the second-stage boot loader.

Note: This document reflects information available at the time of publication. To ensure that you have the most recent information about enhancements to the tools and tool flow, refer to the Intel Arria 10 SoC FPGA Authentication Signing Utility.

Related Information

- Intel Arria 10 SoC FPGA Authentication Signing Utility
- Building Bootloader

Prerequisites

- Supported development platforms:
  - Ubuntu 18.04 or later
  - Windows versions as supported by the Intel Quartus® Prime software
- Intel Quartus Prime Pro Edition and Intel Quartus Prime Standard Edition

Related Information

- Building Bootloader
- Intel Arria 10 SoC Secure Boot User Guide
- Intel Arria 10 Hard Processor System Technical Reference Manual

References

To make the best use of this document, become familiar with the *Intel Arria 10 SoC Boot User Guide*, the *Intel Arria 10 Hard Processor System Technical Reference Manual*, and the *Intel FPGA SoC Embedded Design Suite User Guide*.

Related Information

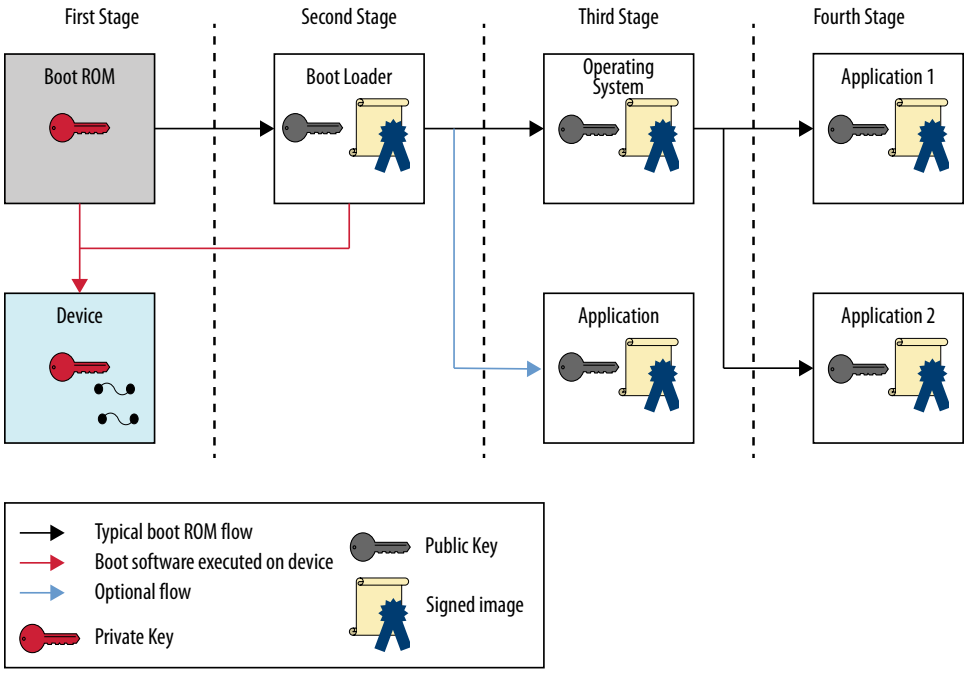
- Intel Arria 10 SoC Boot User Guide
- Intel Arria 10 Hard Processor System Technical Reference Manual
- Intel SoC FPGA Embedded Design Suite User Guide

Secure Boot Stages

A secure boot system ensures that software running on the Intel Arria 10 SoC hard processor system (HPS) is trusted. To ensure this trust, after power-on reset, the HPS executes the trusted first stage boot ROM firmware stored in the device. Each subsequent stage is only loaded and executed if it is authenticated by the current boot stage.

Figure 1. Intel Arria 10 SoC Secure Boot Stages

Note: You can configure the Intel Arria 10 SoC device and the second-stage boot loader so that first and second stages boot securely. If required, you can generate additional signing keys and encryption keys for images in subsequent stages including the OS and application stage. If a subsequent image requires encryption and the encryption key is embedded in the boot loader, then the boot loader image must also be encrypted using the root AES key.



For more information on the Intel Arria 10 boot stages and second-stage boot loader, refer to the *Intel Arria 10 SoC Boot User Guide*.

Related Information

Intel Arria 10 SoC Boot User Guide

Root of Trust

You must establish the root of trust when creating a secure boot system. The root of trust ensures that the security levels are configured properly and the security keys are protected.

Related Information

Software Image Authentication on page 8
For more information on root of trust

First-Stage Boot Loader (ROM)

After hardware system initialization is complete, the Intel Arria 10 SoC boot ROM firmware decrypts, authenticates, and executes the next boot stage. The boot ROM firmware is the root of trust: the trusted, inherently secure starting point for booting the Intel Arria 10 SoC.

To decrypt and authenticate the next boot stage, the boot ROM firmware performs these tasks:

1. Determine which boot device contains the next boot stage image, the second-stage boot loader
2. Discover the final code signing key (CSK) through a key chain service
3. Use the CSK to authenticate the boot loader image
4. If the boot loader image is encrypted, the boot ROM sends the image to the Configuration Subsystem (CSS) for decryption.
5. If boot loader authentication and decryption is successful, load the boot loader into on-chip RAM and execute it

For details about secure system initialization, refer to "Secure Initialization Overview" in the *SoC Security* chapter of the *Intel Arria 10 Hard Processor System Technical Reference Manual*.

Related Information

Secure Initialization Overview

Second-Stage Boot Loader

The second-stage boot loader performs essential tasks to allow an operating system to start.

The boot loader can perform a number of required and optional tasks, such as:

- Configuring I/Os to enable the memory controller prior to FPGA configuration
- Configuring the FPGA portion of the device
- Accessing a file system in flash memory
- Initializing peripherals

In a secure boot implementation, the second-stage boot loader software executes from HPS on-chip RAM.

Third and Fourth Stages

If the stages following the second-stage boot loader need to be trusted, then you must implement features to support authentication in the third and fourth stage.

During the third boot stage, an operating system (OS) or stand alone application, such as Bare Metal, typically loads from flash storage into memory. During the fourth boot stage, the OS commonly launches secure user level applications.

Intel Arria 10 SoC Secure Boot Architecture

You can implement secure boot using the following modules and features provided by the Intel Arria 10 SoC:

- Security Manager
- Boot ROM
- ECDSA Authentication
- Security Fuses
- AES Decryption Engine
- Security Key Storage

A dedicated Security Manager resides in the HPS. It supervises a secure initialization and boot of the system. The Security Manager determines the level of system security in the device by reading the HPS fuse settings after power-on reset (POR).

After the security level is determined, secure boot resources attempt to load software into HPS flash. The boot ROM supervises this bootstrapping process.

Software Image Authentication

Authentication of the second-stage boot loader software by the Intel Arria 10 SoC device provides confidence that it originates from a trusted source. Digital certificates and public key cryptography offer advanced authentication and privacy that less advanced security resources, such as passwords, cannot provide.

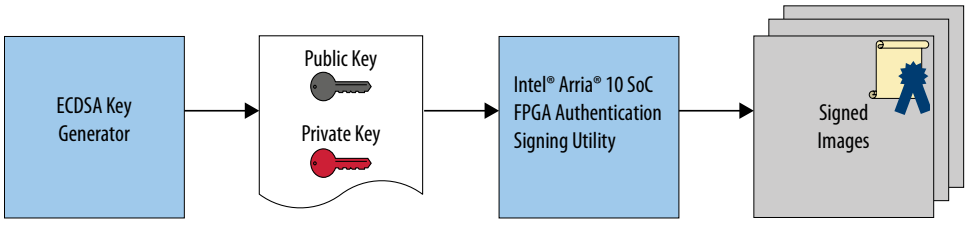
Authentication begins when the boot image is digitally signed. The Intel Arria 10 SoC device requires the image to be signed using an elliptical curve digital signature algorithm (ECDSA) that is based on elliptical curve (EC) cryptography.

Digital Signing

The signing process requires a security key pair and a signing tool to sign the image. The private and public key pair are generated based on a 256 bit ECDSA asymmetric digital signature. The private key has full entropy and is used to derive the public key.

The signing process creates a digital certificate with signatures based on elliptic curve cryptography. The credentials of the signed image during authentication are the digital signature and the public key.

Figure 2. Signing with a Secure Key Pair

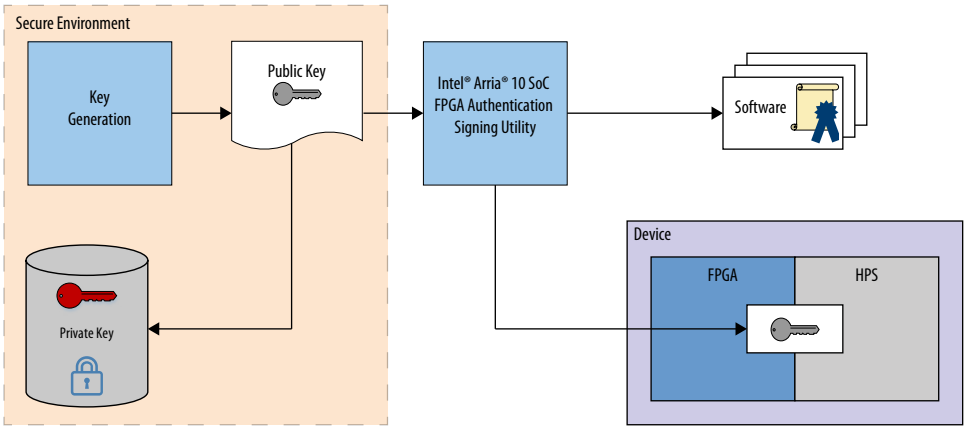


Root of Trust and Root Key

The Root of Trust and the root key pair are the origin where the secure keys are generated. In this secured environment, you can also sign the boot image. A secure environment such as a device manufacturing site, retains the private key to protect it.

The manufacturer generates the root key pair. The root key is programmed into the SoC device and authenticates the software images. The image signing tool is run multiple times for each runtime software on the device. When security is compromised, you must generate a new public key.

Figure 3. Root of Trust



Authentication of the Second-Stage Boot Loader

The security features of the Intel Arria 10 SoC provide you with resources to enforce that only a trusted second-stage boot loader is executed from the HPS. The boot ROM executes the first stage and enforces user security settings. During authentication, the Boot ROM verifies the HPS security fuse settings through the HPS_fusesec shadow registers.

The entire authentication process starts after power-on or cold reset of the device. The process follows a particular order to ensure a secure boot is attempted:

1. On FPGA power-up, the CSS powers, initializes and loads the fuse bits. The CSS sends the FPGA its fuse configuration information. If the HPS is powered, the CSS sends the HPS fuse information to the Security Manager. This information is held in the HPS_fusesec shadow register in the Security Manager.
2. When the Security Manager is released from reset, it requests configuration information from the CSS and performs security checks. At this point, the rest of the HPS is still in reset. The security checks validate whether the state of each security option is valid. The Security Manager decodes the fuse bits and brings the rest of the HPS out of reset.
3. When the HPS is released from reset, the Security Manager sends signals to initialize the system blocks, such as the Clock Manager, FPGA Manager, and System Manager. The clock control fuse information is automatically sent to the Clock Manager, the memory control fuse information is automatically sent to the Reset Manager and all other fuse functions (authentication, encryption, and public

key source and length) are stored in a memory-mapped location for the boot ROM code to read. After these tasks are successfully completed, CPU0 comes out of reset in a secure state.

4. After CPU0 is released from reset, the boot ROM begins executing. At this time, the HPS is in a trusted state and the boot ROM code is guaranteed to execute as expected. For both secure and non-secure boot, all slave peripherals are brought out of reset in a secure state.
5. The boot ROM determines the boot flash partition and verifies the security header settings of the second-stage boot loader image. The second-stage boot loader requires a signed certificate to be authenticated.
6. The Boot ROM determines the source of the root key by reading the security header.
7. The boot ROM attempts to authenticate the boot image. If authentication is successful, the boot ROM then continues with the process of loading and executing the image.

Security Level Staging

After power-on-reset, the Security Manager determines the initial security level by verifying and reading the fuse data. The Security Manager stores the fuse data in the fuse shadow register, HPS_fusesec. From this point, the boot ROM reads the fuse data from the shadow register and also verifies the security header, if present, in the boot image stored on boot flash partition. The second-stage boot loader is the boot image.

The security header may also contain information to raise the security level for a particular feature implemented in the fuses. The boot ROM merges the fuse values in the shadow registers with the security header values to establish the final security level of the system.

Note: Software may program option registers in the Security Manager to raise the security of the system. The higher level of security takes effect immediately and remains at that level until the next cold reset or for some security features, the next warm or cold reset. After reset occurs, the security level returns to the value programmed by the fuse registers and written in the HPS_fusesec registers.

Signed Image

The signing of an image includes prepending an authentication header, including a security header.

Figure 4. Authentication Header

	Final Signature
	Image
Offset to Checksum	Checksum
0x0400	Signatures
0x0240	Spare 448 (0x1C0) bytes
0x0220	Image Data
0x0200	Root Key
0x0140	Spare 192 (0xC0) bytes
0x0100	Option Data
0x0000	Security Header

Figure 5. Security Header

0x002C	Spare 212 (0xD4) bytes
0x0028	Dummy Clocks to Write
0x0020	Date
0x001C	Size after Decryption
0x0018	Flags
0x0014	Offset to Checksum
0x0010	Number of Signatures
0x000C	Load Length
0x0008	Header Length
0x0004	Version (0x00)
0x0000	Validation Word (0x74944592)

Root Key Types

The boot ROM requires the root public key programmed in eFuse and its associated public key to authenticate the second-stage boot loader if the key contained within eFuse, the FPGA or header file (test only) mandates an authenticated flow. Several root key types are available that you can store on the device or second-stage boot loader image.

Note: Using the image itself for storage of the root key is not considered a secure method. Intel recommends that you use this method for testing purposes only.

Table 1. Root Key Types

Root Key	Is it stored on the device?	Description
Secure User Key	Yes	You generate secure key pair for boot ROM to attempt authentication. The SHA256 hash of the public key is stored in the User Access Fuses (UAF) of the device. This configuration provides a secure boot.
FPGA Key	Yes	The public key originates from your bitstream. The key is stored in FPGA on-chip RAM and accessed by the first stage boot ROM for image authentication. When you store the FPGA key in on-chip RAM, you must turn on the **Enable boot from fpga signals** option on the **FPGA Interfaces** tab of the Intel Arria 10 Hard Processor System Intel Arria 10 FPGA IP GUI.
Unsecured User Key	No	You generate a secure key pair but it is not stored on the device. This configuration is considered unsecure. You include the root key result in the image header and the boot ROM uses it for authentication.

Root Public Key Authentication

Before boot ROM can use the root public key for authentication, it must authenticate the root public key against the root public key hash stored in eFuse.

Note: Some key types are unsecure. You can use unsecure keys for testing scenarios where permanent key storage on the device is avoided.

The available key type options are detailed in the *Programming the Secure Signing Key* section.

Related Information

Programming the Secure Signing Key on page 13

Test Secure Boot Authentication

You can perform a secure boot test by using an unsecured key for authentication of the signed boot image. Refer to the *Security Level Staging* section for details of how to increase security on the device.

If you choose to implement the unsecure user key type, then the public key in the signed image is accepted and no check is performed against the SHA256 value stored in the device fuses. You can use this method for testing purposes before you burn the fuses.

Related Information

Security Level Staging on page 10

Programming the Secure Signing Key

After the boot image is signed, the private key is retained in secure storage at the original equipment manufacturer (OEM) to protect it. The public key is programmed into the device. For some signing key types, a hash of the public key is programmed.

The signing key type determines the location of the public key. The available signing key types and corresponding locations are described in the following table.

Table 2. Root Key Types

Root Key	Key Type	Description
Secure User Key	Fuse	You generate secure key pair for boot ROM to attempt authentication. The SHA256 hash of the public key is stored in the User Access Fuses (UAF) of the device. This configuration provides a secure boot. For information about secure fuses, refer to the *Secure Fuses* section in the SoC Security chapter of the *Intel Arria 10 Hard Processor System Technical Reference Manual*.
FPGA Key	FPGA	The public key originates from your bitstream. The key is stored in FPGA on-chip RAM and accessed by the first stage boot ROM for image authentication. When you store the FPGA key in on-chip RAM, you must turn on the **Enable boot from fpga signals** option on the **FPGA Interfaces** tab of the Intel Arria 10 Hard Processor System Intel Arria 10 FPGA IP GUI.
Unsecured User Key	User	You generate a secure key pair but it is not stored on the device. This configuration is unsecure and is for testing only. You include the root key result in the image header and the boot ROM uses it for authentication.

Related Information

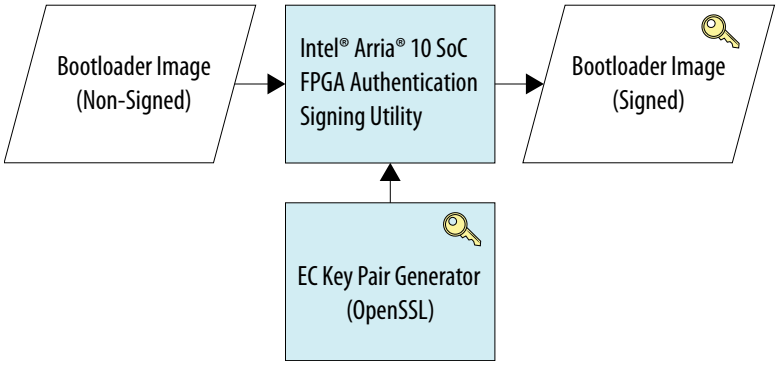
- Secure Boot Stages on page 6
- Generating the Signing Key Pair with OpenSSL on page 14
- Secure Fuses
  For basic information about security fuses, refer to "Secure Fuses" in the *SoC Security* chapter of the *Intel Arria 10 Hard Processor System Technical Reference Manual*.

Boot Image Signing Flow

After you have generated the signing key pair, you can build and sign the boot image with the secure boot image tool.

Figure 6. Boot Image Signing Tool Flow

This diagram illustrates an example tool flow for signing the boot image for authentication.

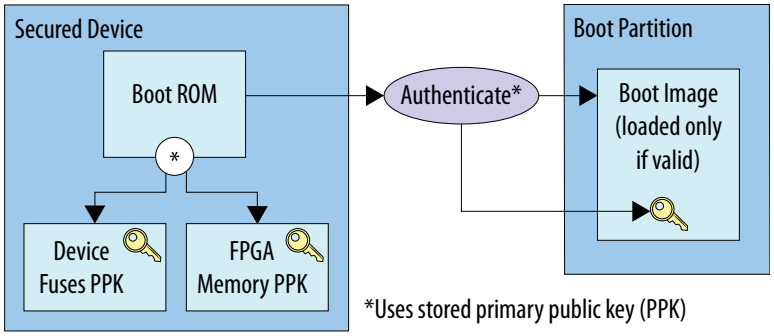


Boot Image Authentication

During a secure boot, the first-stage boot loader (in the boot ROM) uses the root public key and associated key chain to authenticate the second-stage boot loader image as follows:

1. Determine the configuration settings of the device (by reading the fuse values)
2. Attempt to authenticate the boot image, using the root public key type from the configuration settings

Figure 7. Secure Authentication Using Key Types



Related Information

Secure Boot Flow

In the *Booting and Configuration* appendix of the *Intel Arria 10 Hard Processor System Technical Reference Manual*, refer to the following figures: "Verified (Authenticated) Boot Flow", "Second Stage Boot Loader Authentication Process", and "Second Stage Boot Loader Authentication and Decryption Process".

Generating the Signing Key Pair with OpenSSL

You may generate the signing key pair using OpenSSL, an open-source toolkit that supports the Secure Socket Layer (SSL). OpenSSL is available in the Intel Arria 10 SoC FPGA Authentication Signing Utility, and is provided by common Linux distributions.

You invoke OpenSSL from the boot loader generator. OpenSSL applies the security settings that you select in the boot loader generator, and creates an EC key pair. The boot loader generator invokes OpenSSL as follows to generate the key pair:

```
$ openssl ecparam -genkey -name prime256v1 -out root_key.pem
```

In the example above, the generated key pair is stored in the **root_key.pem** file. You can use this file with the Intel secure boot image tool to sign the image.

Related Information

www.openssl.org
Detailed help and information for the OpenSSL toolkit is available on the OpenSSL website.

Overview of the Secure Boot Flow

To create a secure boot system, you can use one of the following secure boot configurations:

- Encrypted only
- Authenticated only
- Encrypted and authenticated

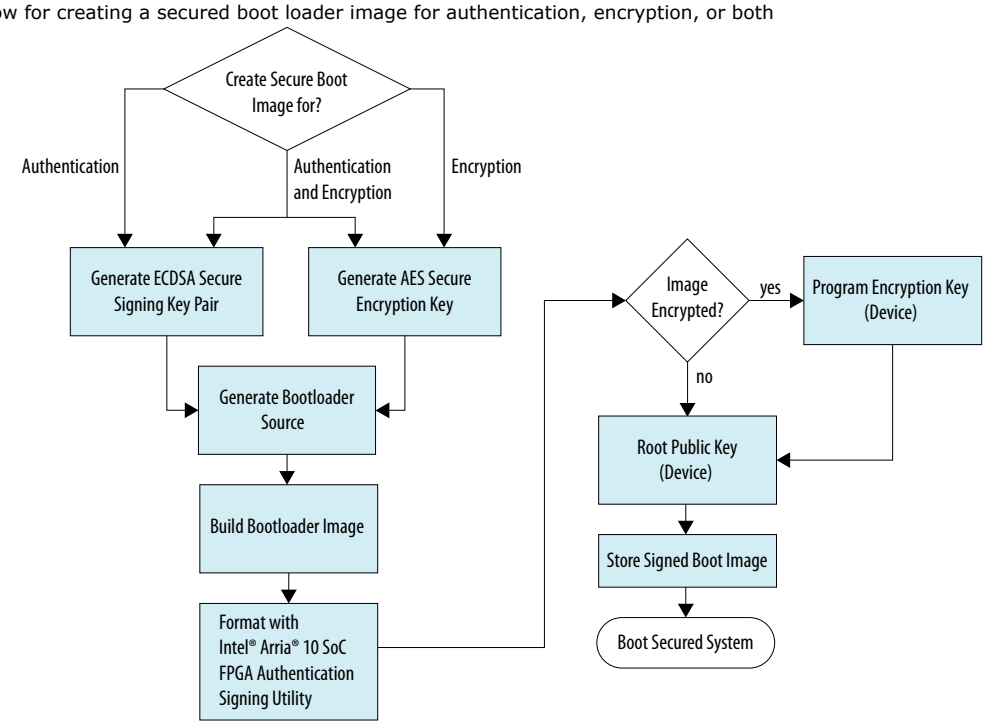
Creating a Secure Boot System

Creating a secure boot loader image entails the following high-level steps:

1. Determine the required security level of the second-stage boot loader: signed for authentication, encrypted, or both.
2. Generate the appropriate secure keys for authentication, encryption, or both.
3. Generate and build the secure boot loader image.
4. Program the secure keys in the Intel Arria 10 SoC device.
5. Configure the security fuses for the desired device security settings.
6. Program the secure boot image to the boot device.

Figure 8. Second-Stage Boot Loader Image Creation Flow

Flow for creating a secured boot loader image for authentication, encryption, or both



Note: To obtain the steps for programming the secure fuses, please contact Intel Support (NDA required).

Related Information

Secure Fuses

For basic information about security fuses, refer to "Secure Fuses" in the *SoC Security* chapter of the *Intel Arria 10 Hard Processor System Technical Reference Manual*.

Software Image Encryption

To encrypt a boot image, you generate and apply encryption keys.

Refer to the "Secure Boot Stages" figure for an overview of key usage.

Related Information

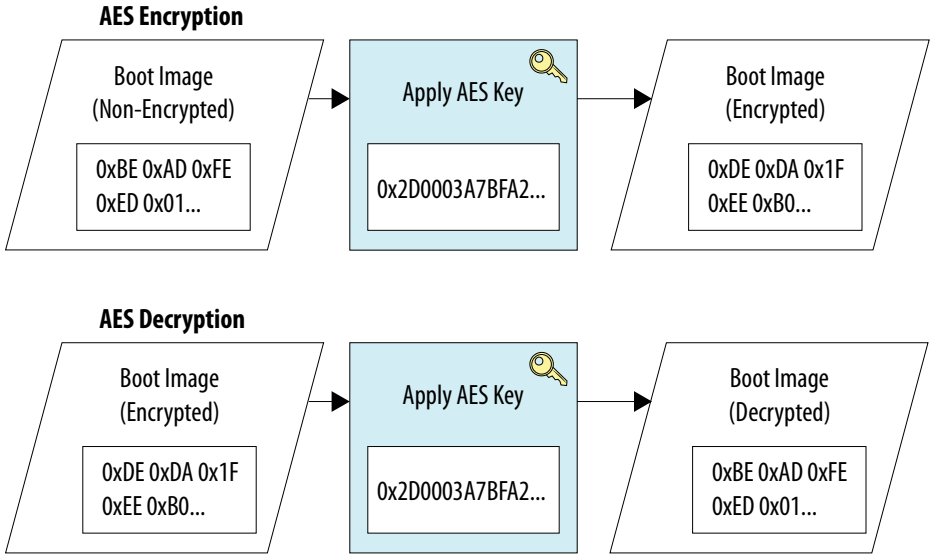
Secure Boot Stages on page 6

AES Encryption and Decryption

The Intel Arria 10 SoC device family supports secure boot with Advanced Encryption Standard (AES) encryption with a 256 bit key length. AES is a symmetric-key algorithm. AES decryption support is provided by the CSS in the FPGA portion of the device. AES decryption is enabled through user fuse settings and software programming.

For information about the CSS, refer to the *SoC Security* chapter in the *Intel Arria 10 Hard Processor System Technical Reference Manual*.

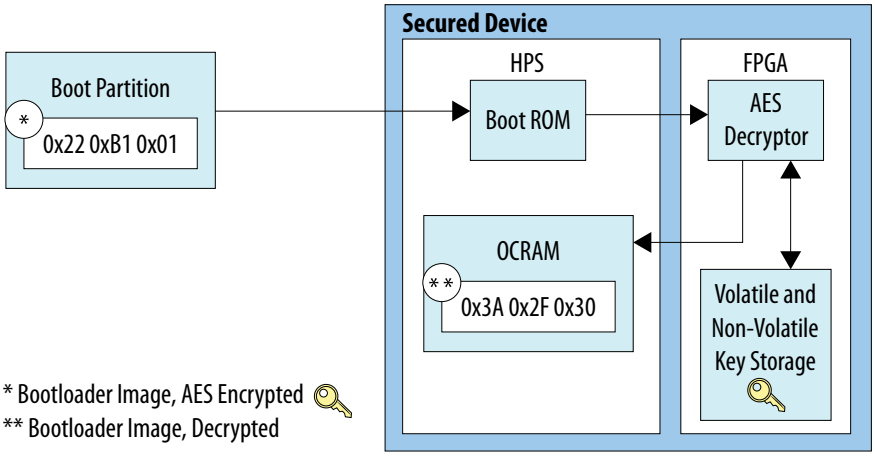
Figure 9. AES Encryption and Decryption



The FPGA portion of the secured device has a dedicated decryption block that uses the AES algorithm to decrypt the boot loader image with a 256 bit AES key that you define. Before receiving the encrypted data, you must write the 256 bit key that you define into the device.

The AES algorithm is a symmetrical block cipher that encrypts and decrypts data in blocks of 256 bits. The decryption block uses the AES algorithm to decrypt the boot loader image and configuration data before configuring the FPGA portion of the device. If encryption is not used, the AES decryptor is bypassed.

Figure 10. Encrypted Second-Stage Boot Loader and the AES Decryptor



Related Information

- SoC Security
  Chapter in the *Intel Arria 10 Hard Processor System Technical Reference Manual*
- Security Encryption Algorithm
  Refer to "Security Encryption Algorithm" in *AN 556: Using the Design Security Features in Intel FPGAs*

Encrypting the Boot Image and Configuration File

The Quartus Prime Design Suite includes the Quartus Prime Convert Programming File tool, **quartus_cpf**, which you use to generate the AES 256 encryption file.(1) You invoke the Quartus Prime Convert Programming File tool as follows:

```
quartus_cpf -e -k <keyfile>:<key_id>[:<key_id>] <input_sof_file>
<output_ekp_file>
```

If you configure the boot loader generator to encrypt the boot image, **quartus_cpf** requires the encryption key file as specified in the configuration tool's security settings. For an overview of the tool flow, see the figure in "Software Image Authentication and Encryption".

For details of Quartus Prime Convert Programming File tool usage, refer to "How to Generate the Single-Device .ekp File and Encrypt Configuration File Using Quartus Prime Software with the Command-Line Interface" in AN-556: *Using the Design Security Features in the Altera FPGAs*.

Related Information

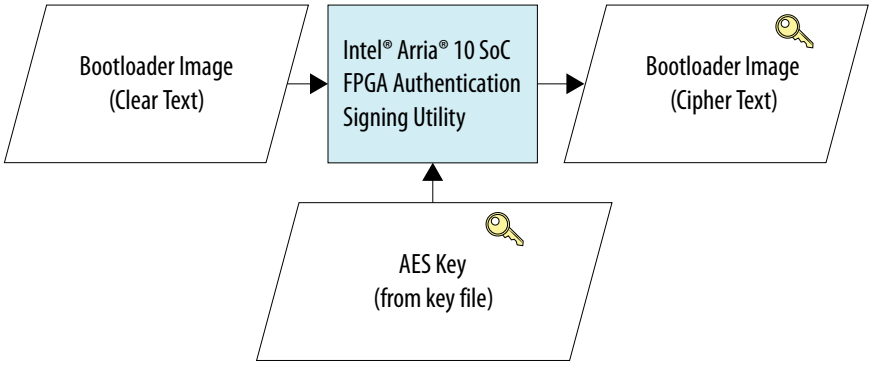
- Software Image Authentication and Encryption on page 21
- How to Generate the Single-Device .ekp File and Encrypt Configuration File Using Intel Quartus Prime Software with the Command-Line Interface
  In *AN 556: Using the Design Security Features in Intel FPGAs*

(1) **quartus_cpf** can also encrypt the configuration bit stream in the SRAM object file (**.sof**).

Boot Image Encryption Flow

Figure 11. Boot Image Encryption Flow

The tool flow for generating an encrypted boot image



Programming the AES Encryption Key

The FPGA device provides both volatile and non-volatile key storage. After the encryption key is generated, you store the key, as described in "Creating an Encrypted Second-State Boot Loader Image". The key is later referenced by the AES-based algorithms that decrypt the boot image. See the "AES Decryption" figure in "AES Encryption and Decryption".

Related Information

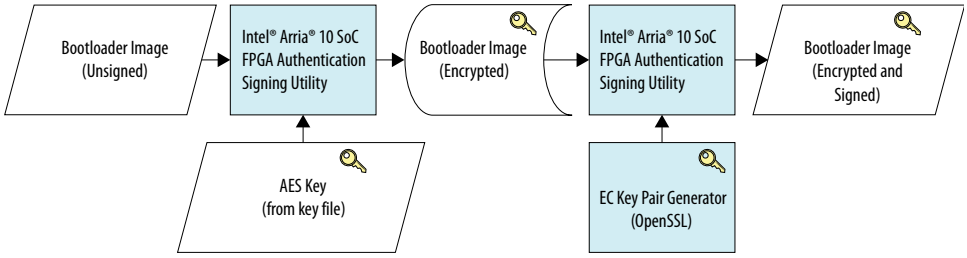
- AES Encryption and Decryption on page 18
- Secure Boot Flow
  Refer to the "Second-Stage Boot Loader Decryption Process" figure in "Secure Boot Flow" in the *Booting and Configuration* appendix to the *Intel Arria 10 Hard Processor System Technical Reference Manual*.
- Authentication and Decryption
  In the *SoC Security* chapter of the *Intel Arria 10 Hard Processor System Technical Reference Manual*

Software Image Authentication and Encryption

To provide the highest level of security during boot, you can apply both signing and encryption to a newly generated second-stage boot loader image. The image must be encrypted first, and then signed, so that the signature is available prior to decryption. During the boot process, the boot ROM firmware first attempts to authenticate the boot loader image. If authentication is successful, the device decrypts and loads the boot loader image.

You can use security settings in the boot loader generator to sign and encrypt a boot loader image.

Figure 12. Boot Image Signing and Encryption Flow



Intel Arria 10 SoC FPGA Authentication Signing Utility

The Intel Arria 10 SoC FPGA Authentication Signing Utility includes tools for creating a secured second-stage boot loader image.

Table 3. Secure Boot Tools

Tool	Name	Description
Secure boot image Python script	`alt_authtool.py`	Python script tool for image signing or encrypting

Secure Boot Image Tool

The Intel Arria 10 SoC FPGA Authentication Signing Utility, `alt_authtool.py`, applies the security settings to the second-stage boot loader image.

If the boot loader is to be authenticated, the secure boot image tool signs the boot loader image with the private key from the previously-generated key pair file. The boot loader generator invokes the tool with the `sign` option and associated parameters from the security settings, as follows:

```
$  python -B -E alt_authtool.py sign [<param1> <param2> ...]
$ python -B -E alt_authtool.py encrypt -k key_file.key:key1 -i u-boot-
mkimage.bin -o uboot-encrypted.abin
```

If the boot loader is to be encrypted, the secure boot image tool encrypts the boot loader image with the key from the previously-generated AES key file. The boot loader generator invokes the tool with the `encrypt` option and associated parameters from the security settings, as follows:

```
$ python -B -E alt_authtool.py encrypt [<param1> <param2> ...]
$ python -B -E alt_authtool.py encrypt -k key_file.key:key1 -i u-boot-
mkimage.bin -o uboot-encrypted.abin
```

Related Information

- Generating the Signing Key Pair with OpenSSL on page 14
- Encrypting the Boot Image and Configuration File on page 19
- Appendix A: Secure Boot Image Python Script: alt_authtool.py on page 29
  Descriptions of all tools used in the secure boot system examples
- Intel SoC FPGA Embedded Design Suite User Guide

Boot Image Format Tool

When you are developing a secure boot loader, you use the boot image format tool to combine up to four boot images to be stored in flash or FPGA memory.

The Intel Arria 10 SoC boot ROM firmware supports up to four boot loader images in flash or FPGA memory, as described in the *Intel Arria 10 SoC Boot User Guide*. When you create a secure boot loader, you must perform an extra step to combine multiple boot loader image files into a single image file.

The boot image format tool formats the boot loader image after it is built. You invoke this tool from the Linux* terminal as follows:

```
$cat <input_image> <input_image> <input_image> <input_image> >\
<output_image>
cat u-boot_w_dtb-signed-256KB.abin u-boot_w_dtb-signed-256KB.abin\
u-boot_w_dtb-signed-256KB.abin\
u-boot_w_dtb-signed-256KB.abin > u-boot_w_dtb-signed-x4.abin
```

The input files are `.bin` or `.abin` files that are typically generated by the boot loader generator. The output file is also a `.bin` or `.abin` file.

Related Information

- Creating a Secure Boot System on page 16
- Intel SoC FPGA Embedded Design Suite User Guide

Secure Boot Examples

You can create a secure boot loader image for authentication, encryption, or both. *Creating a Signed First-Stage Boot Loader Image* and *Creating an Encrypted First-Stage Boot Loader Image* show examples of these processes.

Prerequisites

Python and OpenSSL installation

- Python 3 and modules installation:
  - Install Python 3.6.9 or later and define it as default:

```
sudo apt install python3-pip
sudo ln -s /usr/bin/python3 /usr/bin/python
```

  - Python modules `pyasn1` (0.4.8) and `pynasn1_modules` (0.2.8) for Debian/Ubuntu:

```
pip3 install pyasn1 pyasn1_modules
```

    or

```
pip install --upgrade -r requirements.txt
```

  - Open SSL installation:

```
sudo apt-get install libssl-dev
```

For Windows User:

- Download Python3 from `https://www.python.org/downloads/release/python-363/`
- Install GIT for windows from `https://gitforwindows.org/`
- Add the GIT MinGW binary path to the PATH Environment so that you can use the openssl `(C:\Program Files\Git\mingw64\bin)`

  Note: Make sure you have already installed Intel Quartus Prime Software before using this tool.

Creating a Signed First-Stage Boot Loader Image

The following example shows how to perform the following tasks:

- Create a secure signing key for boot loader image authentication, with the user signing key type.
- Generate and build a boot loader image with the secure signing key, using the Intel Arria 10 SoC FPGA Authentication Signing Utility.
- Demonstrate secure boot using the signed boot loader image from SD card.

Compile hardware design

1. Create top folder:

```
mkdir a10_secure_boot_sign
cd a10_secure_boot_sign
export TOP_FOLDER=`pwd`
```

2. Bring a copy of the hardware design that is already compiled and remove the software folder:

```
cd $TOP_FOLDER
rm -rf a10_soc_devkit_ghrd ghrd-socfpga-ACDS-20.4pro-20.1std\
ACDS-20.4pro-20.1std.zip
wget https://github.com/altera-opensource/ghrd-socfpga/archive/\
ACDS-20.4pro-20.1std.zip
unzip ACDS-20.4pro-20.1std.zip
mv ghrd-socfpga-ACDS-20.4pro-20.1std/\
a10_soc_devkit_ghrd_pro a10_soc_devkit_ghrd
rm -rf ghrd-socfpga-ACDS-20.4pro-20.1std ACDS-20.4pro-20.1std.zip
cd a10_soc_devkit_ghrd
make clean && make scrub_clean && rm -rf software
~/intelFPGA_pro/20.4/nios2eds/nios2_command_shell.sh \
make generate_from_tcl
~/intelFPGA_pro/20.4/nios2eds/nios2_command_shell.sh \
make rbf
```

Build U-Boot

1. Retrieve the U-Boot source code by cloning the git tree and checking out the supported branch:

```
cd $TOP_FOLDER/a10_soc_devkit_ghrd/
mkdir -p software/bootloader
cd software/bootloader/
git clone https://github.com/altera-opensource/u-boot-socfpga
cd u-boot-socfpga
git checkout -b test-bootloader -t origin/socfpga_v2020.10
```

2. Convert `hps.xml` handoff file to include the file to be used by the device tree:

```
cd $TOP_FOLDER/a10_soc_devkit_ghrd/software/\
bootloader/u-boot-socfpga
./arch/arm/mach-socfpga/qts-filter-a10.sh \
../../../hps_isw_handoff/hps.xml \
arch/arm/dts/socfpga_arria10_socdk_sdmmc_handoff.h
```

3. Configure and build U-Boot.

   Note: For the toolchain setup used to build U-Boot, refer to: Building Bootloader web page on RocketBoards.

```
cd $TOP_FOLDER/a10_soc_devkit_ghrd/software/\
bootloader/u-boot-socfpga
export CROSS_COMPILE=arm-none-linux-gnueabihf-
make socfpga_arria10_defconfig
make -j 24
```

Authenticate Image

Retrieve the Secure Boot tools by cloning the git tree:

```
cd $TOP_FOLDER/a10_soc_devkit_ghrd/software/bootloader/
mkdir secure_boot_tools
cd secure_boot_tools
git clone https://github.com/altera-opensource/\
alt-secure-boot
```

Generate Key Pair

```
cd $TOP_FOLDER/a10_soc_devkit_ghrd/software/\
bootloader/
openssl ecparam -genkey -name prime256v1 -out root_key.pem
```

The key file contents should be similar to the following:

```
openssl ec -in root_key.pem  -noout -text
read EC key
Private-Key: (256 bit)
priv:
    ef:a4:fa:45:d9:9d:f8:27:06:f3:d6:48:9b:e3:aa:
    f5:f7:e6:0f:bb:ee:51:44:fe:7b:fb:3f:2a:02:64:
    65:1b
pub:
    04:e5:13:8f:7d:df:0b:83:8c:35:b5:5d:64:3f:f3:
    b6:7a:2f:31:b9:5b:4f:5c:d3:55:18:2a:fa:c1:08:
    e7:be:7e:aa:b0:c8:e7:ea:93:71:3a:62:20:69:3e:
    59:50:60:6e:18:65:d9:a9:95:4c:1e:88:93:06:33:
    b8:cc:05:26:aa
ASN1 OID: prime256v1
NIST CURVE: P-256
```

1. Authenticate image:

```
cd $TOP_FOLDER/a10_soc_devkit_ghrd/software/bootloader/
ln -s u-boot-socfpga/spl/u-boot-spl-dtb.bin

./u-boot-socfpga/tools/mkimage -T socfpgaimage_v1 -d\
u-boot-spl-dtb.bin u-boot_w_dtb-single-mkimage.bin

python -B -E secure_boot_tools/alt-secure-boot/bin/\
alt_authtool.py sign -t user -k root_key.pem -i\
u-boot_w_dtb-single-mkimage.bin -o u-boot_w_dtb-signed.abin
```

The key file contents should be similar to the following:

```
openssl ec -in root_key.pem  -noout -text
read EC key
Private-Key: (256 bit)
priv:
    ef:a4:fa:45:d9:9d:f8:27:06:f3:d6:48:9b:e3:aa:
    f5:f7:e6:0f:bb:ee:51:44:fe:7b:fb:3f:2a:02:64:
    65:1b
pub:
    04:e5:13:8f:7d:df:0b:83:8c:35:b5:5d:64:3f:f3:
    b6:7a:2f:31:b9:5b:4f:5c:d3:55:18:2a:fa:c1:08:
    e7:be:7e:aa:b0:c8:e7:ea:93:71:3a:62:20:69:3e:
    59:50:60:6e:18:65:d9:a9:95:4c:1e:88:93:06:33:
    b8:cc:05:26:aa
ASN1 OID: prime256v1
NIST CURVE: P-256
```

2. After running the Python script with the Intel Arria 10 SoC FPGA Authentication Signing Utility, expect a SHA256 digest similar to the one below:

```
SHA256 digest of root public key:
233c42a16266942910d801bf717006148fc869bf96027ef76e478731d59e3a6d
```

3. Generate the four copies of the image in one file:

   Note: Refer to your toolchain directory where you get the cross compiler.

```
../../../../Toolchain/gcc-arm-10.2-2020.11-x86_64\
-arm-none-linux-gnueabihf/bin/arm-none-linux-gnueabihf-objcopy\
-I binary -O binary --gap-fill 0x00 --pad-to 0x40000\
u-boot_w_dtb-signed.abin u-boot_w_dtb-signed-256KB.abin

cat u-boot_w_dtb-signed-256KB.abin u-boot_w_dtb-signed-256KB.abin\
u-boot_w_dtb-signed-256KB.abin u-boot_w_dtb-signed-256KB.abin >\
u-boot_w_dtb-signed-x4.abin
```

4. Since we are using user mode, program the authentication key file (`root_key.pem`) into the board (virtually), because it is part of the image.
5. Copy the `u-boot_w_dtb-signed-x4.abin` to the board flash:
   - SD/MMC—Use the A2 (raw) partition

     For more information about where to place this image, refer to the *Intel Arria 10 SoC - Boot from SD Card* section on RocketBoards.
   - QSPI
   - NAND
6. Boot the board.

Related Information

- Programming the Secure Signing Key on page 13
- Intel Arria 10 SoC - Boot from SD Card section on RocketBoards
- Creating an Encrypted First-Stage Boot Loader Image on page 26
- Second Stage Bootloader Support Package Generator
- BSP Generator Graphical User Interface
  In the *Intel SoC FPGA Embedded Design Suite User Guide*: detailed information about creating a boot loader

Creating an Encrypted First-Stage Boot Loader Image

The following example demonstrates how to perform the following tasks:

- Create a secure encryption key for boot loader image authentication.
- Generate and build an encrypted boot loader image with the secure encryption key, using the Intel Arria 10 SoC FPGA Authentication Signing Utility.
- Demonstrate secure boot using the encrypted boot loader image from SD card

1. Follow steps Step 1 and Step 2 for Bootloader generation as explained in the *Creating a Signed First-Stage Boot Loader Image* section.
2. Retrieve the Secure Boot tools by cloning the git tree.

```
cd $TOP_FOLDER/a10_soc_devkit_ghrd/\
software/bootloader/
mkdir secure_boot_tools
```

```
cd secure_boot_tools
git clone https://github.com/altera-opensource/\
alt-secure-boot
```

3. Generate Encryption Key—Create a new file named as `key_file.key`, give a name to the encrypt key say "key1" and give a random 32 bytes of hex number to the key1 the contain looks like this: key1 0a0b0c0d0e0f112233445566778899a0b0c0d0e0f112233445566778899aabb

```
touch key_file.key
echo key1 0a0b0c0d0e0f112233445566778899a0b0c0d0e0f\
112233445566778899aabb > key_file.key
```

4. Create `ubootmkimage`:

```
create u-boot_w_dtb-single-mkimage.bin
mkimage -T socfpgaimage_v1 -d  u-boot-spl-dtb.bin u-boot_w_dtb-single-
mkimage.bin
```

5. Encrypt image

```
cd $TOP_FOLDER/a10_soc_devkit_ghrd/software/bootloader/
ln -s u-boot-socfpga/spl/u-boot-spl-dtb.bin

./u-boot-socfpga/tools/mkimage -T socfpgaimage_v1 -d\
u-boot-spl-dtb.bin u-boot_w_dtb-single-mkimage.bin

~/intelFPGA_pro/20.4/nios2eds/nios2_command_shell.sh

python -B -E secure_boot_tools/alt-secure-boot/bin/alt_authtool.py\
encrypt -k key_file.key:key1 -i u-boot_w_dtb-single-mkimage.bin\
-o u-boot_w_dtb-encrypted.abin
```

6. Generate the four copies of the image in one file:

   Note: Refer to your toolchain directory where you get the cross compiler.

```
../../../../Toolchain/gcc-arm-10.2-2020.11-x86_64\
-arm-none-linux-gnueabihf/bin/arm-none-linux-gnueabihf-objcopy\
-I binary -O binary --gap-fill 0x00 --pad-to 0x40000\
u-boot_w_dtb-encrypted.abin u-boot_w_dtb-encrypted-256KB.abin

cat u-boot_w_dtb-encrypted-256KB.abin u-boot_w_dtb-encrypted-256KB.abin\
u-boot_w_dtb-encrypted-256KB.abin u-boot_w_dtb-encrypted-256KB.abin\
> u-boot_w_dtb-encrypted-x4.abin
```

7. Program the Encryption key file (`key_file.key`) into the BBRAM on the board.

```
quartus_pgm --key "key_file.key:key1"\
ghrd_10as066n2.sof key.ekp
quartus_pgm -c 1 -m jtag -o p;key.ekp,10AS066H2ES -o
```

8. Copy the `u-boot_w_dtb-signed-x4.abin` to the board flash:
   - SD/MMC—Use the A2 (raw) partition

     For more information about where to place this image, refer to the *Intel Arria 10 SoC - Boot from SD Card* section on RocketBoards.
   - QSPI
   - NAND
9. Boot the board.

Related Information

- Intel Arria 10 SoC - Boot from SD Card section on RocketBoards
- Creating a Signed First-Stage Boot Loader Image on page 23
  Boot loader generator dialog box
- How to Generate the Single-Device .ekp File and Encrypt Configuration File Using Intel Quartus Prime Software with the Command-Line Interface
  In *AN 556: Using the Design Security Features in Intel FPGAs*
- Steps for Implementing a Secure Configuration Flow
  In *AN 556: Using the Design Security Features in Intel FPGAs*
- The *SD Card Boot Utility* chapter of the *Intel SoC FPGA Embedded Design Suite User Guide*

Creating a Signed and Encrypted First-Stage Boot Loader Image

The following example shows how to perform the following tasks.

1. Create a secure signing and an encryption key.
2. Generate and build an encrypted boot loader image with the secure encryption key, using the Intel Arria 10 SoC FPGA Authentication Signing Utility.
3. Demonstrate secure boot using the encrypted boot loader image from the SD card.
4. Follow steps Step 1 to Step 4 from the *Creating an Encrypted First-Stage Boot Loader Image* section.
5. Use the encrypted image in Step 1 and sign the image, refer to the steps from the *Creating a Signed First-Stage Boot Loader Image* from Step 3.
6. Program the authentication key and encryption key to the board virtually.
7. Copy `u-boot_w_dtb-signed-encrypted-x4.abin` to the SD/MMC, QSPI, and NAND flash on the board.
8. Boot the board.

Appendix A: Secure Boot Image Python Script: `alt_authtool.py`

Example 1. Secure Boot Image Tool Usage for Boot Image Authentication (Signing)

```
python -E -B alt_authtool.py --help
usage:
  python -E -B alt_authtool.py sign [-h] \
    --inputfile INPUTFILE --outputfile OUTPUTFILE \
    [--fuseout FUSEOUT] [--pubkeyout PUBKEYOUT] \
    [--rootkey-type {fuse,fpga,user}] \
    [--keypair KEYPAIR] \
    [--fpga-key-offset FPGA_KEY_OFFSET]

Sign a bootloader image to allow BootROM verification

optional arguments:
  -h, --help            show this help message and exit
  --inputfile INPUTFILE, -i INPUTFILE
                        Bootloader image to sign
  --outputfile OUTPUTFILE, -o OUTPUTFILE
                        Signed output image
  --fuseout FUSEOUT, -fo FUSEOUT
                        Hash of root public key, to be burned into device
                        fuses
  --pubkeyout PUBKEYOUT, -pko PUBKEYOUT
                        Root public key in raw data form. This data may then
                        be built into the FPGA image for usage with
                        --rootkey-type=fpga
  --rootkey-type {fuse,fpga,user}, -t {fuse,fpga,user}
                        The trusted root key's type. (default: fuse) 'fuse':
                        embed root pubkey in image. BootROM verifies its hash
                        against device fuses. 'fpga': fetch trusted root
                        pubkey from location in FPGA memory. 'user': embed
                        root pubkey in image. BootROM does not verify.
  --keypair KEYPAIR, -k KEYPAIR
                        Signature keypairs specified in order from the
                        trusted root key to final user key
  --fpga-key-offset FPGA_KEY_OFFSET
                        Offset from H2F bridge base address (0xC0000000) to
                        location of logic-embedded root public key. Used for
                        '--rootkey-type fpga' authentication.
```

Example 2. Secure Boot Image Tool Usage for Boot Image Encryption

```
python -E -B alt_authtool.py encrypt --help
usage:
  python -E -B alt_authtool.py encrypt [-h] \
    --inputfile INPUTFILE --outputfile OUTPUTFILE \
    --key KEY [--non-volatile]

Convert a pimage into an encrypted boot image

optional arguments:
  -h, --help            show this help message and exit
  --inputfile INPUTFILE, -i INPUTFILE
                        Bootloader image to encrypt
  --outputfile OUTPUTFILE, -o OUTPUTFILE
                        Encrypted output image
  --key KEY, -k KEY     File containing symmetric key to use for encryption
  --non-volatile        Decryption key stored in non-volatile fuses, instead
                        of battery-backed storage
```

Appendix B: Frequently Asked Questions

Table 4. Frequently Asked Questions (FAQs) Summary Table

Topic
What are the secure configurations for HPS JTAG debug and access? on page 31
Can the HPS perform decryption of the boot image instead of the FPGA CSS? on page 31
What happens if the first stage boot ROM is unsuccessful in authenticating the second-stage boot loader? on page 31
Can you use the first-stage root key as the subsequent stage root key? on page 31
When the second-stage image is authenticated, is the image header only copied to on-chip RAM for authentication? on page 32
Can the AES encryption key be updated by the HPS using JTAG hosting? on page 32
How does U-Boot (SSBL) authenticate next stage boot images? on page 32
Which elliptical cryptography is used for boot image signing and authentication? on page 32
How do I generate a signing key pair? on page 32
Where can I store the signing keys for second-stage boot loader authentication? on page 32
What type of cryptography is used for boot image encryption and decryption? on page 33
What FPGA locations are available for AES key storage? on page 33
How do I generate an AES key to encrypt a boot image? on page 33
How is secure boot defined within the Intel Arria 10 SoC product family? on page 34
What security choices are available for the second-stage boot image or user software? on page 34
Where is the authentication of the boot image performed? on page 34
How can I configure the Intel Arria 10 SoC device so that it always performs authentication or authentication and decryption? on page 34
How can I program the key authentication key (KAK) into the Intel Arria 10 SoC device? on page 35
How can I configure the second stage boot loader image for the correct authentication signing key type? on page 35
How do I configure the second-stage boot loader image for encryption using the pre-generated AES key? on page 35
Is the ECDSA private and public key pair that is used for signing the boot image also used for authentication of the FPGA image? on page 35

What are the secure configurations for HPS JTAG debug and access?

Two efuse bits, dbg_disable_access and dbg_lock_JTAG, control the secure JTAG debug configurations. You can read the programmed efuse values for your device through the HPS_fusesec register. A bit value of 1 in the HPS_fusesec register represents a "blown" fuse state and a 0 represents an "unblown" fuse state.

The table below describes the possible HPS configurations with JTAG. The dbg_access_f and dbg_lock_JTAG columns reflect the efuse value of these bits in the HPS_fusesec register. If both efuse are unblown then after the device exits reset, full JTAG access is possible. This configuration is the default configuration.

Table 5. JTAG Security Configuration Options

JTAG Configuration	dbg_disable_access	dbg_lock_JTAG	Description
HPS JTAG include	0	1	• This configuration includes the HPS in the JTAG chain by default. • Your software application cannot remove the HPS from the JTAG chain. • This configuration allows HPS debug from power-on reset.
HPS JTAG exclude	1	1	Permanently exclude the HPS from the JTAG chain.
Default	0	0	Enable JTAG with software debug programmability.

Can the HPS perform decryption of the boot image instead of the FPGA CSS?

The HPS portion of the SoC does not support AES operations. It can only perform public key-based authentication. The HPS can, however, push the boot image into the FPGA CSS and perform the same decryption used in the FPGA configuration flow.

When decryption is complete, the CSS returns the image to the HPS and the HPS uses that image as the boot image. The HPS and FPGA share the same AES root key which is stored in efuse. The CSS uses a simple key derivation function, AES (efuse or BBRAM key, #constant) for the HPS and FPGA configuration images.

What happens if the first stage boot ROM is unsuccessful in authenticating the second-stage boot loader?

The first stage boot ROM attempts to authenticate all four second stage images that are stored in the boot partitions of your flash device. If the device cannot authenticate the images or identifies the images as corrupt, then the boot ROM attempts to execute a fall back image located in the on-chip RAM of the FPGA.

Can you use the first-stage root key as the subsequent stage root key?

Intel recommends using a separate final signing key between different boot stages. Intel does not recommend using a root key for the first-stage or subsequent stage boot loader direct signing. Sharing the same root key between the first-stage and subsequent stage boot loader is only successful if you use the same ECC algorithm for each.

When the second-stage image is authenticated, is the image header only copied to on-chip RAM for authentication?

The entire boot loader image is always copied into the on-chip RAM and authenticated.

Can the AES encryption key be updated by the HPS using JTAG hosting?

You can only update the AES key in volatile memory through a connected JTAG interface. The HPS does not support JTAG hosting.

How does U-Boot (SSBL) authenticate next stage boot images?

The current GSRD U-Boot does not feature image authentication beyond the second stage bootloader (U-Boot). You can enable U-Boot to authenticate subsequent boot images (Linux) by configuring or adding authentication capability to U-Boot. Reference the latest U-Boot releases for support on authentication. You may also want to add specific third-party or open source solutions.

Which elliptical cryptography is used for boot image signing and authentication?

The Intel Arria 10 SoC device family uses the elliptical curve digital signing algorithm with NIST-approved ECDSA on NIST P-256 curve for signing and authentication of second-stage boot images.

How do I generate a signing key pair?

You may use the open source OpenSSL toolkit or your own tool to generate a key pair file that contains a private and public key pair. The Intel Arria 10 SoC FPGA Authentication Signing Utility boot tool requires a key pair file for signing an image. If you decide to use OpenSSL, refer to the OpenSSL website for more information about how to use the tool.

Related Information

www.openssl.org
Detailed help and information for the OpenSSL toolkit is available on the OpenSSL website.

Where can I store the signing keys for second-stage boot loader authentication?

You can store the signing keys for second-stage boot loader authentication by the Intel Arria 10 SoC device in:

Table 6. Root Key Types

Root Key	Key Type	Description
Secure User Key	Fuse	You generate secure key pair for boot ROM to attempt authentication. The SHA256 hash of the public key is stored in the User Access Fuses (UAF) of the device. This configuration provides a secure boot.

continued...

Root Key	Key Type	Description
		For information about secure fuses, refer to the *Secure Fuses* section in the SoC Security chapter of the *Intel Arria 10 Hard Processor System Technical Reference Manual*.
FPGA Key	FPGA	The public key originates from your bitstream. The key is stored in FPGA on-chip RAM and accessed by the first stage boot ROM for image authentication. When you store the FPGA key in on-chip RAM, you must turn on the **Enable boot from fpga signals** option on the **FPGA Interfaces** tab of the Intel Arria 10 Hard Processor System Intel Arria 10 FPGA IP GUI.
Unsecured User Key	User	You generate a secure key pair but it is not stored on the device. This configuration is unsecure and is for testing only. You include the root key result in the image header and the boot ROM uses it for authentication.

Related Information

Secure Fuses
For basic information about security fuses, refer to "Secure Fuses" in the *SoC Security* chapter of the *Intel Arria 10 Hard Processor System Technical Reference Manual*.

What type of cryptography is used for boot image encryption and decryption?

The Intel Arria 10 SoC device family supports secure boot using the Advanced Encryption Standard (AES) encryption with a 256 bit key length. You can encrypt your boot image using quartus_cpf tools. The Intel Arria 10 SoC AES engine only supports decryption.

What FPGA locations are available for AES key storage?

Within the FPGA, you can store the public (root) key in key registers located in:

- User fuses (non-volatile memory)
- Battery-backed RAM (volatile memory) within the FPGA

The contents of the volatile key registers are retained between power-cycles with battery power. Non-volatile key registers are fuse-based and are one-time programmable.

How do I generate an AES key to encrypt a boot image?

The AES key file (`.key`) is a text file that you generate using a true random number generator (TRNG) or some other trusted tool. Refer to *AN 556: Using the Design Security Features in Intel FPGAs* for the content format of this file.

Related Information

- Encrypting the Boot Image and Configuration File on page 19
  For more information about using `quartus_cpf` to store keys
- AN 556: Using the Design Security Features in Intel FPGAs
  For content format of the AES key file

How is secure boot defined within the Intel Arria 10 SoC product family?

Within the Intel Arria 10 SoC device family, a secure boot implies that before the system loads any user (non-device modifiable) software, such as a second-stage boot loader image, it:

- Checks the image for authenticity
- Decrypts any encrypted image before signing it as certified

What security choices are available for the second-stage boot image or user software?

Authentication is provided for the second-stage boot loader code and both the HPS and FPGA can utilize the AES algorithms in the CSS to decrypt boot images and POF files, respectively.

Three levels of boot are available to the device:

- Authentication only: The second-stage boot loader code is not encrypted, but there are public key signatures attached to the image and the code only executes if all of the signatures pass. ECDSA256 (SHA 256) is used for authenticated boot.
- Decryption only: The user boot code is encrypted and must be decrypted before execution. AES-based algorithms in the FPGA are used for decryption.
- Authentication and Decryption: The user boot code is encrypted and signed.

If authentication and decryption are enabled, the data is first authenticated and then decrypted using the AES algorithms. Authentication is performed using the public key authorization key (KAK) held in the user fuses. The KAK can be 256 or 512 bits. You can lock the KAK public key authentication fuses in groups of 64 bits or less.

Where is the authentication of the boot image performed?

The HPS boot ROM authenticates the boot image in the SoC. The FPGA does not perform this authentication.

Where is decryption of the boot image performed?

If the boot ROM detects that the boot image is encrypted, it sends the image to the CSS for the AES to perform decryption.

How can I configure the Intel Arria 10 SoC device so that it always performs authentication or authentication and decryption?

You can ensure that the Intel Arria 10 SoC device always performs a signed authentication check or an authentication check with runtime decryption by programming the device fuses for these features and by using the required security keys. Specifically, you must:

- Program the **aes_en_f** fuse so that an AES decryption of a flash image is always performed.
- Program the **kak_src_f** fuse to indicate where the key authorization key (KAK) resides.
- Program the **kak_len_f** fuse to configure the length of the KAK.
- Program the **authen_en_f** fuse so that HPS authentication is required for all flash images prior to execution.
- Program the security authorization key in the location you have selected.

How can I program the key authentication key (KAK) into the Intel Arria 10 SoC device?

You can program the KAK into the device fuses permanently using the Intel FPGA Download Cable and the programmer tool installed with the Design Tool Suite.

How can I configure the second stage boot loader image for the correct authentication signing key type?

You must select the appropriate security settings for authentication before generating the second-stage boot loader in the Intel Arria 10 SoC FPGA Authentication Signing Utility. After the settings are applied, you build the boot loader and the configurations are incorporated in the image. After these steps, you must build and sign the boot loader.

If you use the GIT repo to build the boot loader source, then you must build the image and then use `alt_authtool.py` to sign the final image.

How do I configure the second-stage boot loader image for encryption using the pre-generated AES key?

If you require a signed and encrypted second-stage boot loader image for authentication and decryption, then the image is encrypted prior to signing. Otherwise the image is encrypted after the source is generated and the image is built. You encrypt the final image using the Intel Arria 10 SoC FPGA Authentication Signing Utility, `alt_authtool.py`. You must select the appropriate security settings for encryption before generating the second-stage boot loader in the `alt_authtool.py`. After the settings are applied, you must build the boot loader image to include the configuration.

Is the ECDSA private and public key pair that is used for signing the boot image also used for authentication of the FPGA image?

The ECDSA signing key pair is only used for signing of the second-stage boot image. The FPGA does not support public key-based authentication.

Document Revision History for the AN 759: Using Secure Boot in Intel Arria 10 SoC Devices

Document Version	Changes
2021.03.29	• Removed information about bsp-editor. • Removed information about Intel SoC FPGA Embedded Design Suite (EDS). • Removed *Boot Loader Generator*.
2019.11.08	Updated the definition of FPGA Key to include the following information: When you store the FPGA key in on-chip RAM, you must turn on the **Enable boot from fpga signals** option on the **FPGA Interfaces** tab of the Intel Arria 10 Hard Processor System Intel Arria 10 FPGA IP GUI.
2017.11.06	• Updated "Secure Boot Stages" figure in *Secure Boot Stages* section to include more stage details. • Added Third and Fourth Stages on page 7 subsection to the Secure Boot Stages topic. • Clarified authentication process in *Software Image Authentication* section and added the subsections: — Digital Signing on page 8 — Root of Trust and Root Key on page 9 — Authentication of the Second-Stage Boot Loader on page 9 — Security Level Staging on page 10 — Signed Image on page 11 — Root Key Types on page 12 — Root Public Key Authentication on page 12 • Added the *Appendix B: Frequently Asked Questions* section.
2016.03.29	Initial release